m2103

TM (4) Projet calculette : arbres

Table des matières

1 Présentation

Dans cette partie, on voit

- comment représenter des expressions par des structures arborescentes définies inductivement (on combine des constantes, des variables, etc. par des opérations) ;
- comment implémenter ces structures en Java ;
- comment effectuer des traitement dessus (récursivement) ;

Pour étudier ces notions plus à l'aise, on travaillera sur un projet séparé (à télécharger le moment venu).

À terme l'objectif sera d'intégrer la construction des arbres à la calculette, pour laquelle vous savez déjà analyser les expressions.

2 Les expressions sont des arbres

En mathématiques, la tradition est d'utiliser une notation à 1 ou 2 dimensions pour écrire les expressions. Les deux expressions ci-dessous sont équivalentes

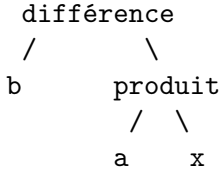
```
                        b - a * x
(b - a * x)/(c + d)     ---------
                          c + d
```

on y remarque l'utilisation

- de **conventions de priorité** entre opérateurs, par exemple dans `b - a*x`,
- de **parenthèses** pour former des sous-expressions,

qui font que, par exemple, on sait que `b-a*x` est équivalent à `b-(a*x)`, mais pas à `(b-a)*x`.

La même expression peut se représenter d'une infinité de façons, en ajoutant des parenthèses superflues à volonté : `((((b)) -((a))*(x)))`. Mais fondamentalement, il s'agit de la *différence* entre la *variable b*, et le *produit de a et x*. Ce qu'on peut dessiner sous forme d'un **arbre** :



On dit que cet arbre est une "syntaxe abstraite" de l'expression, par définition à la syntaxe concrète, la représentation textuelle linéaire.

Pour bien comprendre

1. Dans l'arbre ci-dessus, on remplace "différence" par "quotient", et "produit" par "somme" ; écrivez l'expression correspondante.
2. Dans `a-b+c`, on emploie deux opérateurs de même priorité. Comment grouper ?
3. Dessinez l'arbre pour `a-b*c-d+e*f`.

3 Les arbres

Les arbres peuvent être définis *inductivement* :

Un arbre est composé de noeuds qui sont :

- soit une **feuille** contenant une variable,
- soit un **noeud binaire** qui représente la composition de **sous-expressions** par une **opération** (addition, produit, …). Ces sous-expressions sont elles-mêmes codées par des arbres.

Dans l'exemple, on a seulement des opérations binaires sur des variables, mais il n'est pas dificile d'imaginer qu'on peut ajouter :

- des feuilles qui contiennent des **constantes**
- de noeuds qui correspondent à des opérations **unaires** (valeur absolue, opposé, …)

Travail dessinez un diagramme de classes.

4 Codage des arbres

Les différents types de noeuds sont implémentés par des classes qui ont une interface commune.

Le code qui utilise les noeuds n'aura pas besoin de connaitre leur type exact (polymorphisme) pour leur appliquer des traitements

4.1 Les opérations applicables aux noeuds

A priori, on demandera à un noeud

- de calculer et retourner sa `valeur`
- de fournir une description de ce qu'il représente. Par exemple le texte `la constante 2 PLUS la variable a`.

4.2 Le modèle de données

- interface `Expr`
- implémentée par les classes :
  - `ExprConstante`, contenant une valeur ;
  - `ExprVariable`, contenant le nom de la variable ;
  - `ExprBinaire`, avec une opération et les noeuds des sous-expressions (parties gauche et droite)

Les types d'opérations binaires seront recensés par une énumération

```
enum OpBinaire {  PLUS, MOINS, ... }   ...
```

Variante : On pourrait imaginer avoir une classe pour chaque opération (`ExprSomme`, `ExprDifference`, ….), mais cela entraîne un risque de prolifération de classes au code très similaire.

4.3 L'encapsulation

On va regrouper ce qui concerne les noeuds dans un `package`. Le but d'un `package` est de fournir de fonctionnalités, tout en cachant au maximum les détails de réalisation.

Concrètement, le programmeur utilisateur du package a besoin d'un moyen pour construire des arbres, qui correspondent à différents types d'expressions (constantes, variables, somme etc.)

Cela ne veut pas dire qu'on **doit** lui donner accès à une multiplicité de classes. Ce dont il a besoin en réalité, c'est de méthodes qui fabriquent des noeuds de différentes sortes. Pas d'appeler `new`.

En pratique, on fait appel à des méthodes statiques de création

```
Expr e =     Expr.binaire(                    // ici
                        Expr.constante(3),    // et ici
                        OpBinaire.PLUS,
                        Expr.variable("x")    // ici aussi
                        ) ;
```

qui sont définies dans l'interface `Expr` :

```
public interface Expr {

    public static Expr constante(int valeur) {
        return new ExprConstante(valeur) ;
    }
    // ...
}
```

La logique est l'implémentation des différentes sortes d'expressions par des classes `ExprConstante` etc. est un choix technique de réalisation, qui concerne pas l'utilisateur du package.

On limite donc la visibilité des classes au package.

```
class ExprConstante implements Expr {   // accès "package"

    private final int valeur ;

    ExprConstante(int valeur) {  // constructeur : accès "package"
        this.valeur = valeur ;
    }
    ...
```

4.4 Le calcul de la valeur,

Comme une expression peut contenir des variables, la notion de "valeur d'une expression" n'a de sens que si des valeurs sont attribuées aux variables.

Exemple : si a=4 et b=1, la valeur de "3*a + b" est 13 .

L'évaluation d'une `Expr` se fait donc par rapport à un objet, l'`Environnement`, qui lui fournit la valeur des variables

```
interface Expr {
    int valeur(Environnement env) ;
    ...
}
```

4.5 L'Environnement

Un environnement est une simple association entre des noms (de variables) et des valeurs entières

```
public interface Environnement {
    void affecter(String name, int value) ;
    int  valeur(String name) ;
}
```

Une implémentation `TableVariables` est fournie (tableau associatif).

5 Travail

5.1 Préparation

- Installer le Projet Netbeans fourni (Arbres.zip)

5.2 Faire tourner

Faites tourner le programme. Il s'affichera

```
Valeur de la constante 3 = 3
Valeur de la constante 3 PLUS la constante 4 = 7
Valeur de la constante 10 MOINS la constante 2 PLUS la constante 5 = -3
```

Les deux premières lignes correspondent (dans `Arbres.java` aux expressions 3 et 3+4. Ce qui s'affiche parait correct.

5.3 Corrections

Pour la troisième, il y a un problème.

- l'intention est d'évaluer l'expression qui correspond à 10-(2+5).
- la description qui s'affiche `la constante 10 MOINS la constante 2 PLUS la constante 5` n'est pas satisfaisante. Elle donne l'impression qu'il s'agit de `10-2+5`

Votre premier travail est de trouver ce qu'il faut modifier pour "parenthéser" les expressions binaires. Et de le faire. On voudrait voir quelque chose du genre :

```
Valeur de (la constante 10 MOINS (la constante 2 PLUS la constante 5)) = ...
```

Votre second travail est de remarquer qu'il y a une erreur de calcul, d'en trouver la cause, et de la corriger.

5.4 Extension

Dans `testsSansVariables`, il faut écrire le codage de l'expression `(2+3)*4`, et faites afficher sa description et sa valeur. Pour cela, vous devrez ajouter la multiplication.

5.5 Traitement des variables

Dans la méthode `testsAvecVariables`, on met affecte "à la main" des valeurs à des variables `a`, `b` et `c`.

Pour pouvoir tester l'évaluation des expressions `a`, `a+b`, `a+b*c`, il faut pouvoir les construire. Exemple

```
afficherAvecValeur(Expr.variable("a"), vars) ;
```

et pour cela écrire une méthode statique `Expr.variable` qui fabrique et retourner une `ExprVariable` (à définir).

Les tests suivants devraient vous convaincre que ça marche (ou remarquer que ça ne marche pas).

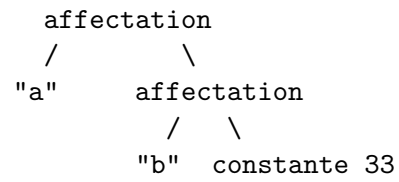
5.6 Traitement de l'affectation

Dans la calculette, on a choisi de ranger l'affectation parmi les expressions, c'est-à-dire qu'une affectation retourne une valeur. Ce qui permet

- de les chainer comme dans `a = (b = 33)`
- de les traiter comme les expressions, puisque *ce sont* des expressions.

Mais elles comportent - un nom de variable - une expression. là où les expressions binaires avaient une opération et deux expressions.

Il faut donc introduire un nouveau type de noeud



et donc une nouvelle implémentation de `Expr`.

Travail

1. Ecrivez le test pour la première affectation

```
// a = 10
afficherAvecValeur(Expr.affectation("a", Expr.constante(10)) ;
```

2. Ecrivez la méthode de construction pour qu'elle crée une instance de `ExprAffectation`

```
class ExprAffectation implement Expr {
    ExprAffectation(String nom, Expr expr) {
       ...
    }
}
```

3. Complétez les méthodes `d'ExprAffectation`, et vérifiez le fonctionnement sur les autres tests.

6 Conclusion

6.1 Synthèse

1. Les expressions arithmétiques peuvent être représentées par des objets arborescents, définis par **induction**. On combine des objets de base (constantes, variables) pour faire des objets de plus en plus gros.
2. L'évaluation (et d'autres traitements, comme la production d'une description) s'exprime naturellement de façon **récursive** : la valeur d'une expression qui est une différence s'obtient en soustrayant les valeurs des sous-expressions.
3. Pour la réalisation on a employé un "**type composite**" : une interface (ou une classe abstraite) d'un type T, dont les implémentations peuvent avoir des attributs qui contiennent eux-mêmes des éléments de type T. C'est un "patron de conception" très courant (Exemple dans un système de fichiers : un élément est soit un fichier, soit un répertoire qui contient des éléments, …)

- L'emploi de méthodes statiques pour construire les objets composites permet de cacher ces détails techniques sous le tapis.

6.2 Prochaine étape.

Ici on a vu comment construire un arbre qui correspond à une expression, et comment l'évaluer.

Dans les exemples, les arbres étaient construits laborieusement, à la main.

L'étape suivante est d'employer l'algorithme d'analyse de la calculette pour contruire l'arbre d'une expression.

Vous vous rappelez qu'on avait quelque chose du genre

```
int evaluerExpr() {
    int valeur = evaluerTerme();
    while (token.isSymbol("+")) {
        nextToken();
        int tmp = evaluerTerme();
        valeur = valeur + tmp;
    }
    return valeur;
}
```

ça ne va pas plus compliqué, on suivra exactement le même schéma pour produire une `Expr` plutôt qu'un entier :

```
Expr arbreExpr() {
    Expr expr = arbreTerme();
    while (token.isSymbol("+")) {
        nextToken();
        Expr tmp = arbreTerme();
        expr = Expr.binaire(expr, OpBinaire.PLUS, tmp);
    }
    return expr;
}
```

Le traitement d'une ligne par la calculette consistera alors à

1. analyser la ligne pour produire l'arbre de l'expression,
2. l'afficher,
3. afficher sa valeur.